ISI/TM-77-6
October 1977

ARPA ORDER NO. 2223

PRIM System:

- U1050 User Guide
- User Reference Manual

Louis Gallenson
Alvin Cooperband
Joel Goldberg

INFORMATION SCIENCES INSTITUTE
UNIVERSITY OF SOUTHERN CALIFORNIA
4676 Admiralty Way/ Marina del Rey/ California 90291
(213) 822-1511

DISCLAIMER NOTICE

THIS DOCUMENT IS BEST QUALITY PRACTICABLE. THE COPY FURNISHED TO DDC CONTAINED A SIGNIFICANT NUMBER OF PAGES WHICH DO NOT REPRODUCE LEGIBLY.

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE	READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER: ISI/TM-77-6	2. GOVT ACCESSION NO. / 3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle): PRIM System: U1050 User Guide and User Reference Manual	5. TYPE OF REPORT & PERIOD COVERED: Technical manual
	6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s): Louis Gallenson, Alvin Cooperband, Joel Goldberg	8. CONTRACT OR GRANT NUMBER(s): DAHC15-72-C-0308 ARPA Order 2223
9. PERFORMING ORGANIZATION NAME AND ADDRESS: Information Sciences Institute, 4676 Admiralty Way, Marina del Rey, CA 90291	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS: ARPA Order #2223, Program Code 3D30 & 3P10
11. CONTROLLING OFFICE NAME AND ADDRESS: Defense Advanced Research Projects Agency, 1400 Wilson Blvd., Arlington, VA 22209	12. REPORT DATE: Oct [redacted] 77 / 13. NUMBER OF PAGES: 49
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office): 49p.	15. SECURITY CLASS. (of this report): Unclassified / 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

This document approved for public release and sale; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

debugging tool, emulated I/O, emulation-based programming tools, emulators, microprogramming

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

This is a two-part manual for users of the PRIM-based U1050 emulator. The manual demonstrates as well as describes the capabilities of PRIM, running and debugging of object code, and the emulated computer system.

407 952

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE S/N 0102-014-6601

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

ISI/TM-77-6
October 1977

ARPA ORDER NO. 2223

PRIM System:

- U1050 User Guide
- User Reference Manual

Louis Gallenson
Alvin Cooperband
Joel Goldberg

INFORMATION SCIENCES INSTITUTE

UNIVERSITY OF SOUTHERN CALIFORNIA

4676 Admiralty Way/ Marina del Rey/ California 90291
(213) 822-1511

THIS RESEARCH IS SUPPORTED BY THE ADVANCED RESEARCH PROJECTS AGENCY UNDER CONTRACT NO. DAHC15 72 C 0308, ARPA ORDER NO. 2223, PROGRAM CODE NO. 3D30 AND 3P10.

CONTENTS

User Reference Manual

PRIM SYSTEM: U1050 USER GUIDE

INTRODUCTION

The PRIM system supports U1050 program development and testing by providing an emulated U1050 tool embedded in an interactive time-sharing environment. This emulated U1050 provides powerful debugging aids not found on an actual U1050 computer system.

This guide consists of two sections, serving distinct purposes. This first section is an extended introduction to the PRIM U1050 tool and its capabilities; it is addressed to the U1050 user with no prior exposure to PRIM. It consists of an overview of the tool, followed by a detailed discussion of a number of common or representative programming problems with solutions illustrated by means of transcripts of actual sessions with the PRIM U1050 emulation tool. The second section is an appendix to a separate document, ***The PRIM System: User Reference Manual***; that manual and the appendix together constitute the complete reference document for the PRIM U1050 tool. (The PRIM system supports a family of emulation tools; the ***User Reference Manual*** covers the capabilities of the PRIM system as they apply to all the tools in general.)

PRIM is available through both the NSW (National Software Works) and the USC-ISIC TENEX system, which is a server system on the ARPANET. The user of PRIM is assumed to have access to one or the other system and some rudimentary familiarity with its use. Once PRIM has been entered, its behavior is identical in the two environments.

The PRIM U1050 tool consists of the emulated U1050 plus two separate command interpreters, known as the exec and the debugger. At any time, PRIM is either running the U1050 emulator or processing user commands to the exec or debugger; the transition between these states is at the control of the user.

Exec commands are concerned primarily with the manipulation of U1050 environments and configurations. The elements of exec commands are keywords, file names, and (decimal) numbers. Keywords include such items as command names, device names, options, and parameters. They need not be entered in their entirety; any unambiguous leading substring of the desired word suffices for recognition. (When a keyword is terminated with an *escape* character, the word is completed by the exec.) File names refer to files in the user's file space (in either NSW or TENEX), and follow the appropriate file name syntax. Each file specification requires the name, as appropriate, of either an existing file (to be read or modified) or a new file (to be created and written).

Debugger commands are concerned with the detailed control of the emulated U1050. The debugger includes the functions available on the front panel of the U1050 as a small subset of its capabilities. Each debugger command word consists of a single character; the arguments to debugger commands are symbolic expressions involving the elements of the U1050 (e.g., memory locations, index registers, CC, or tetrads), user-defined names (e.g., program labels or debugger formats), or constants.

Within PRIM, certain ASCII characters have been assigned special functions when input by the user. These functions, which are described completely in the ***User Reference Manual***, concern command editing and PRIM (command and U1050) control. The command

editing functions are backspace (either the ASCII *backspace* or *cntl-A* characters), backup (*cntl-W*), delete (*del* or *rubout*), retype (*cntl-R*), and question (*question-mark*); the control functions are status (*cntl-S*) and abort (*cntl-X*). Backspace backs up over one character within the current field of a command; it is acknowledged by a backslash (\) followed by the erased character. Backup backs up over the current command field; it is acknowledged by a backslash (\) followed by the first character of the erased field. Delete backs out of the current subcommand entirely (or out of the current command if not in a subcommand); it is acknowledged by "XXX", followed by a new prompt. Retype re-displays the current command or subcommand line. Question, when entered at the beginning of a field, generates a summary of the input currently expected, followed by a retype of the line.

Status causes PRIM to respond with the current status of the emulated U1050. Abort causes any operation in progress to be terminated cleanly and returns control to the top level of PRIM (to either the exec or debugger, depending on which one last had control). The abort function is used both to abort a command that is partially entered or in process and to stop the running U1050.

With this background we can now illustrate how PRIM can be used. Two examples will be explored in detail; these examples will show how to:

1. Key in a small program, run it to discover a bug, fix the bug, trace it to observe its operation, and save the results.

2. Load a program into memory from a card deck, interrupt and save the session, and continue it later from the same point.

Several further examples will be presented with considerably less detail to illustrate a number of ways in which the interactive PRIM debugger can be used. These examples will show how to:

1. Find which instructions are modifying a location and which ones are setting it to a designated value.

2. Determine how many times a code sequence is executed and how many times a data location is referenced prior to a known condition occurring.

3. Trace a loop only once.

4. Find which instructions in a program were *not* executed.

5. Test if a data location has changed over a code sequence.

6. Search a buffer for a given value.

In general, these examples will consist of transcripts taken from PRIM sessions. The transcripts typically will be embedded within a paragraph of text that guides the reader through the example and explains or amplifies selected features of it.

ENTER AND DEBUG A SMALL PROGRAM

To start, let us follow a complete step-by-step sequence of interactions in running PRIM, keying in a sample program, running and debugging it, and saving the patched result. In this and all the remaining examples of this guide, user inputs are in bold italics with serifs, whereas PRIM responses are in a light sans-serif font. Input control characters are represented as superscripts.

The session will start at the TENEX exec level (the prompt character is "@"), where the U1050 emulator will be called in from the PRIM directory (the PRIM prompt character is ">"). A similar step would be taken to run the PRIM U1050 tool under NSW. Then the PRIM debugger will be called in (the prompt character is "#") so that a small program can be keyed in. The debugger includes an assembler that can recognize U1050 assembly-language instructions and generate the binary equivalent.

```
@<PRIMesc>beiU1escBSO.SAV; 6 cr
U1050 ( 4/02/77)
>DescEBUG
#Mode Instruction esc #Type 01000esc
 01000: 00 = FT 0164,44\
 01005: 00 = PD0 013600+6,7\
 01012: 00 = SC 013606,033\
 01017: 00 = PD 013477,4\
 01024: 00 = XF 021,,,3\
 01031: 00 = XF 040,02000,,3\
 01036: 00 = XF ,02000,,3\
 01043: 00 = JC $+10,43\
 01050: 00 = J $-10\
 01055: 00 = FT 0161,44\
 01062: 00 = XF 022,,,3\
 01067: 00 = XF 040,02000,,3\
 01074: 00 = XF ,02000,,3\
 01101: 00 = JC $+10,43\
 01106: 00 = J $-10\
 01113: 00 = JD $+5\
 01120: 00 = J 01067cr
#Go (to) 01000cr
--> Unimplemented instruction at 00: 00   #
```

The program was keyed in by typing location 01000 (in instruction mode) and specifying replacement (ending the command with an *escape* character rather that a *carriage-return*). The first instruction of the test program was entered as the replacement value, terminated by a *back-slash* ("\") to force a display of the next available location so that it, too, could be replaced. The last new instruction was terminated by a *carriage-return*, which completed the replacement sequence. (At that point, the user might have reentered the type-with-replacement command so as to inspect the program, instruction by instruction, and correct any input errors.) The user executed the program by telling the debugger to start at (go to) location 01000. This command transferred control from the PRIM debugger to the emulator, which executed the program until it encountered a condition that caused it to stop execution -- in this case, an illegal instruction at location 0.

To determine how the program got to location 0, the user will request a history display of the last 32 jumps and find that the last jump occurred from location 01036, an

XF instruction. Since this instruction is not a program jump, the transfer of control must have been caused by an interrupt, so the state of the interrupt-inhibit flops (*IR1*, *IR2*, and *IR3*) will be examined, showing that a class-3 interrupt occurred, presumably because the one-second one-shot triggered by the instruction at location 01024 had fired. Because the channel-3 interrupt vector at location 0435 was never set, such an interrupt would vector to location 0. This will be corrected by entering an interrupt-return instruction at location 0430 and the vector to it at location 0435. After checking the change, the "CLEAR button" will be "pressed" (using the set command) and the program will be reexecuted.

```
#Jump-history cr
01050--01036(31 times)   01036--00   #Type IR1:IR3cr
 IR1: 00    IR2: 00    IR3: 01   #Mode Instruction esc #Type 0430esc
 0430: 0104300 = J $cr
#Type 0435:+3esc    0435: 00 = cr
 0436: 00 = 4,030cr
#Type M435cr
 M435: 04300000   #Set S.CLEARcr
#Go (to) 01000cr
--> Halt display at 01113: JD 01120
--> U1050 halted at 01120, Used 0:03.0
#Type DAY.CLOCKcr
 DAY.CLOCK: 030303030305   #
```

Note on the second line how the interrupt-inhibit flops were specified as a range of locations, from *IR1* through *IR3*, and on the fifth line how locations 0435-0437 were specified as a range-start and an increment. Note on the seventh line also how the five-byte sequence of addresses from 0435-0441 was specified as the 30-bit-wide cell *M435*. This time when the program was run it stopped where expected. The contents of the day-time clock were examined and found to have been stepped twice since being set to zero.

The program now appears to cycle, but to verify that successive interrupts are being handed properly, the debugger will be instructed to break execution on every interrupt and the program will be continued.

```
#Break (at) INTcr
#Go (to) cr
--> Halt display at 01113: JD 01120
--> U1050 halted at 01120, Used 0:04.0
#Type DAY.CLOCKcr
 DAY.CLOCK: 030303030306   #
```

Instead of breaking, however, the program halted again at the JD instruction with the clock having advanced once more, without an interrupt having occurred.

The program will be cycled one more time and found still not to generate an interrupt. The interrupt-inhibit flops will be examined, showing that class-3 interrupts are inhibited. Realizing that the interrupt-return instruction in location 0430 should have reset *IR3*, the user will modify it to clear the inhibit flop. Then the System Clear "pushbutton" will be set and the program restarted.

```
#Go (to) cr
--> Halt display at 01113: JD 01120
--> U1050 halted at 01120, Used 0:05.0
#Type DAY.CLOCKcr
 DAY.CLOCK: 030303030307   #Type IR1:IR3cr
 IR1: 00    IR2: 00    IR3: 01   #Mode Instruction esc ##Type 0430esc
 0430: JU 0430 = JC $,031cr
#Set S.CLEARcr
#Go (to) 01000cr
--> Class 03 interrupt   #Type DAY.CLOCKcr
 DAY.CLOCK: 030303030304   #Type IR3cr
 IR3: 01   #lf
--> Single-step at 0430: JC 01043,031   #:cr
 IR3: 00   #Go (to) cr
--> Class 03 interrupt   #Go (to) cr
--> Halt display at 01113: JD 01120
--> U1050 halted at 01120, Used 0:07.0
#Type DAY.CLOCKcr
 DAY.CLOCK: 030303030305   #Go (to) cr
--> Class 03 interrupt   #
```

This time (on line 10) the interrupt occurred. After examining the clock and *IR3*, the program was single-stepped (by entering *line-feed* as a debugger command) and *IR3* was reexamined to verify that it had been cleared. Note on line 13 that by entering *colon* as a debugger command the last location displayed by the user was redisplayed; type-outs produced by breakpoints do not change what is considred to be the last location displayed. Continuing the program showed that it was interrupting correctly.

The program now appears to work correctly. To demonstrate this, the user will next trace the execution of the program to follow the interrupts as they occur. This will be done by clearing all existing breakpoints and then setting an execute-break on every instruction within the program and at the interrupt-return instruction; to avoid tracing through the two idle loops (instructions at locations 01036-01054 and 01074-01112), the execute breaks set for them will be deleted. In order to produce a trace, a simple break-time "debugger program" will be entered for each breakpoint that is set; by terminating the break command with an *escape* rather than a *carriage-return*, a break-subcommand mode will be entered (an extra prompt character indicates this) wherein debugger commands can be supplied that are to be executed automatically at the time the break condition is detected by the emulator. This break-time program, in effect, will state: "print the contents of the cell pointed to by *OLDCC* as an instruction and continue" (breaks occur *after* the CPU cycle in which the condition is detected, so CC has already been advanced).

```
#Debreak (from) esc all (confirm)cr
#Break (at) 01000:01120, 0430esc (after doing) Xecute esc
 ##Mode Instruction esc ##Type @OLDCCcr
 ##Go (to) cr
 ##cr
<Program number is [1]>   #Debreak (from) 01036:01054, 01074:01112cr
#
```

Note on the second line that the execute-break locations were entered as a list (the range 01000-01120 and the location 0430) and that entering an "empty" subcommand (*carriage-return* by itself) on the fifth line terminated the break-subcommand mode and the break

command. Every new break-time program is assigned a unique sequence number so that it can be designated easily; this number was reported on the sixth line after the command had been accepted. Note also on the sixth line that a list of two ranges was used to remove breakpoints from both idle loops with the same command. And note on the third line that the operator "@" was used to get the value contained in *OLDCC* (the value of CC before it was advanced); as "@" is the normal TIP intercept character, users accessing PRIM via a TIP should redefine the intercept character.

A display of existing breakpoints will be requested to verify that they had been specified correctly. After a system clear, the program will be reexecuted. (It should be noted that "pushing" a console "button" only sets a function bit for the emulator; the implementation of the function does not occur until the emulator is run next.) This time, however, because of the break program, each instruction executed (except those within the idle loops) will be displayed after its execution.

```
#Break (at) cr
0430 <X>   01000-01035 <X>   01055-01073 <X>   01113-01120 <X>   #Set S.CLEARcr
#Co (to) 01000cr
--> 01000: FT 0164,054
--> 01005: PD0 013606,07
--> 01012: SC 013606,033
--> 01017: PD 013477,04
--> 01024: XF 021,00,00,03
--> 01031: XF 040,02000,00,03
--> 0430: JC 01043,031
--> 01055: FT 0161,054
--> 01062: XF 022,00,00,03
--> 01067: XF 040,02000,00,03
--> 0430: JC 01074,031
--> 01113: JD 01120
--> Halt display at 01113: JD 01120
--> U1050 halted at 01120, Used 0:10.0
#Co (to) cr
--> 01120: J 01067
--> 01067: XF 040,02000,00,03
--> 0430: JC 01074,031
--> 01113: JD 01120
--> Halt display at 01113: JD 01120
--> U1050 halted at 01120, Used 0:11.0
#Co (to) cr
--> 01120: J 01067
--> 01067: XF 040,02000,00,03
--> 0430: JC 01074,031
--> 01113: JD 01120
--> Halt display at 01113: JD 01120
--> U1050 halted at 01120, Used 0:12.0
#
```

Continuing the program twice from the JD instruction demonstrated that the interrupts occurred as expected.

To complete the first example, the program will be saved by returning from the PRIM debugger to the PRIM exec, where a save of the U1050 memory into the file CLOCK.TEST will be requested.

```
@Return (to EXEC) cr
>SAesceVE ? One of the following:
 ALL
 CONFIGURATION
 MEMORY
 SYMBOLS
>SAVE Mesc EMORY (on file) CLOCK.TEST;1 cr
>Qesc UIT
  Quitting U1050  (Confirm) cr
@
```

Note on the second line that the *question-mark* elicited the available options within the save command. When the save command finished, PRIM was terminated, returning control again to the TENEX exec (note the prompt character "@" on the last line).

LOAD A CARD DECK FOR EXECUTION

The second detailed example will go through the process of loading a program from a binary card deck. When the U1050 emulator is called in, a standard hardware configuration is supplied. This configuration consists of a U1050 processor with a line printer, card reader, card punch, mass storage unit (disk), file loader (tape), operator's console, and the day-time clock installed. The user's (controlling) terminal is automatically connected to the operator's console. In order to use any other peripheral device, however, it must be connected to an input source and/or output destination -- either to the user's terminal or to a TENEX file (or pair of files). The process of connecting the U1050 device to the TENEX system is called "mounting." Either a TENEX file or the user's terminal is "mounted" on the U1050 device. The day-time clock has arbitrarily been assigned to unit number 16 on channel 3; although it appears in the peripherals display, it is not a mountable device. In the following example, a basic U1050 configuration will get TENEX files mounted; then a card deck will be loaded into memory. The resulting system will be stopped, saved, and continued at a later time.

```
@<PRIMesc>beiUesc050.SAV; 6 cr
U1050 ( 4/02/77)
>Pesc ERIPHERALS
Chan  Unit  Mounted  Device
 0     0       No  PRINTER
 1     0       No  READER
 2     0       No  PUNCH
 3     0      Yes  LS-COMM-UNIT
 3    16       No  CLOCK
 6     0       No  DISK
 7     0       No  TAPE
>Fesc ILESTATUS (for device) cr
 Record   File Name          Device
 0        User Tty           LS-COMM-UNIT
>
```

The configuration displayed with the peripherals command showed that only LS-comm-unit zero (operator's console) was mounted. The status of mounted TENEX files was displayed with the Filestatus command; it showed that the user's terminal was mounted on the LS-comm-unit.

In this example the user wishes to load the program deck identified as 148-71 and *run it. To allow printout from that program,* a file will have to be mounted on the line printer. To load 148-71, a file containing the binary deck will have to be mounted on the card reader. And to run 148-71 successfully, a file representing the disk (as set up by a previous program) will also have to be mounted.

```
>MESCOUNT (A,I,N,OL,OU,T,?) ? One of the following:
 APPEND
 INPUT
 NEW
 OLD
 OUTPUT
 THIS-TERMINAL
>MOUNT (A,I,N,OL,OU,T,?) OUESCTPUT (to file) SCRATCH.PRT; 31CR
(on device) ? One of the following:
 PRINTER
 READER
 PUNCH
 COMM-UNIT
 DISK
 TAPE
 CLOCK
>MOUNT (A,I,N,OL,OU,T,?) OUTPUT (to file) SCRATCH.PRT; 31
(on device) PRESCINTER CR
>
```

The mount command, as with all PRIM exec commands, is largely self-guiding. Whenever the PRIM exec expects input from the user, the input of a *question-mark* will elicit either a list of options or a description of what form of input is expected. In general, a mount command designates what kind of file, the file name, and the U1050 device involved. Note that the *question-marks* entered to the mount command elicited the available options -- the types of file (when entered on line 1 prior to the first argument) and the device names (when entered on line 9 prior to the next argument). An output file, SCRATCH.PRT, was mounted on the printer; it is assumed by PRIM to be an ASCII file.

A binary input file, 148-71.CRD, will be mounted on the card reader; if a translated card deck is expected, either an ASCII or a column-binary file may be used, but if an untranslated deck is expected, only a column-binary file may be used. In this case, the deck for 148-71 already has been loaded onto a TENEX file in column binary (i.e., each column of the card is represented by 12 bits in the file, with the top row being the most-significant bit and the bottom row the least significant bit); each column unit is a 12-bit byte. A binary input/output file, DISK, will be mounted for the disk; the emulator assumes this file to be binary with a 6-bit byte.

```
>Mesc OUNT (A,I,N,OL,OU,T,?) Iesc NPUT (from file) 148-71.CRD; Iesc (on device)
 Resc EADER cr
>>? BINARY or ASCII
>>Besc INARY
>>cr
>Mesc OUNT (A,I,N,OL,OU,T,?) OLesc D (in & out file) DISK.; Iesc (on device)
 Desc ISK cr
>Fesc ILESTATUS (for device) cr
 Record   File Name          Device
 0        SCRATCH.PRT; 31     PRINTER
 0        148-71.CRD; 1       READER
 0        User Tty           LS-COMM-UNIT
 0        LGS.DSK; 1          DISK
>
```

Because the card reader's file can be either ASCII or binary, a subcommand mode was entered for the mount command where file characteristics could be supplied. Since byte size for a binary card reader (or card punch) file is assumed to be 12 bits, the user was not asked to specify it. And since a disk file is assumed to be binary with a 6-bit byte, the user also was not asked to specify it. The filestatus command showed that the printer, reader, operator's console, and disk had been mounted. Note that as no specific device was designated, the status of files was displayed for all mounted devices.

Since the user wants to stop the CPU after the deck has been loaded, a breakpoint will be set at the first instruction of the program in order to stop the emulator so that the session can be saved for future execution starting at a point after the load has occurred. In order to load the card deck, the PRIM debugger must be used to "press" the console switches. After pressing the "Load-card" button (i.e., using the debugger to set the indicator named C.LOAD), the "Start" button will be pressed (with the go-to command.)

```
>Desc EBUG
#Break (at) 01000esc (after doing) Xecute cr
#Set C.LOADcr
#Go (to) cr
--> Break after executing  01000: JR 03666,00   #Debreak (from) esc all
 [confirm]cr
#
```

The breakpoint was removed (with the debreak command) prior to saving the state of the emulation so that it could not interfere with any future use of the saved session. Note on the last line that the default of removing *all* breaks required the *escape* character and a confirmation; this is to prevent inadvertent removal of breakpoints.

An arbitrary amount of processing could occur at this point, with the session being interrupted at any convenient point to be saved and restored at a later time. In order to save the session, control first must return to the PRIM exec.

```
@Return (to EXEC) cr
>SAescVE ? One of the following:
 ALL
 CONFIGURATION
 MEMORY
 SYMBOLS
>SAVE Aesc LL (on file) 148-71.LOADED; 1cr
>Qesc UIT
  Quitting U1050  (Confirm) cr
@
```

The save command permits the user to save just the U1050 system configuration, just the U1050 memory, just user symbol tables, or all of these along with whatever additional information is required to *exactly* reestablish the then-current situation at a later time. (It should be noted, however, that the contents of files are not saved, so that if a read/write file on some device is changed subsequently, then when a save-all is restored the state of the total system cannot be *exactly* recreated.) Although a save-memory would be adequate at this point for most purposes, a save-all was performed onto the file 148-71.LOADED so that existing I/O conditions would be reestablished on subsequent restores. A save-all takes more time to save and restore than does a save-memory and also results in a much larger file, so save-all should be used only where it is necessary to capture the internal state of the CPU and peripherals. The save-all command can be especially useful when a program error requires a great deal of processing to occur before it appears. By saving the session shortly before the error occurs, the error condition can be recreated quickly and easily. Quitting the PRIM exec returned control to the TENEX exec.

At some arbitrary later time the user can again run the U1050 emulator and continue a saved session where he left off at the time it was saved. This will be shown using the restore command, completing the second example.

```
@<PRIMesc,bel U1esc050.SAV; 6 cr
U1050 ( 4/02/77)
>RESescTORE (from SAVE file) 148-71.LOADED; 1cr
  restoring ALL from TUESDAY, FEBRUARY 22, 1977 09:57:15-PST
>Fesc ILESTATUS (for device) Resc EADER
 Record   Type  Byte/Last     File Name          Device
 66       Bin12 5280/5280    148-71.CRD; 1       READER
>Desc EBUG
#Type OLDCC;CCcr
 OLDCC: 01000    CC: 03570   #
```

After the restore of the save-all from file 148-71.LOADED, the filestatus display showed that the card deck had already been read in. Note on the fifth line that when the filestatus display was requested for a specific device (in this case, the reader) additional information was supplied, such as the file type (binary with 12-bit bytes), and the current and last byte position of the file; the activity status also appears when a device is active. After the PRIM debugger was called in, *OLDCC* and *CC* were displayed, showing that the last instruction had executed out of location 01000 and the next one was to be executed out of location 03570. At this point the session can be continued as if it had never been interrupted.

The remaining examples will be much briefer than those presented above. Instead of complete sequences of interaction with an actual program, just those commands that are

necessary to solve particular problems will be shown. In a few cases intervening interactions have been edited out of the transcript to emphasize the essential commands. In general, results will not be shown.

FIND WHICH INSTRUCTIONS MODIFY A LOCATION

A typical debugging problem is to find what instructions are changing a location (e.g., some module is clobbering a parameter). This can be solved very easily with a simple break-time debugger "program". If the contents of location 012345 are being changed (and should not be), the following breakpoint command will identify the culprit.

```
#Break (at) 012345esc (after doing) Write esc
 ##Mode Instruction esc ###Type @OLDCCcr
 ##Type 012345cr
 ##Co (to) cr
 ##cr
<Program number is [1]>  #
```

After every write reference to location 012345, program execution will be interrupted and control will be passed to the debugger, which will execute the break-time commands to display the instruction that just executed and to display location 012345. In this example the program would continue automatically after each break. If only the first breakpoint subcommand had been entered (eliminating the type-out of 012345 and the go commands on the third and fourth lines), then after displaying the last instruction executed, the debugger would display location 012345 (with its new contents) and pass control to the user.

FIND WHICH INSTRUCTION SETS A LOCATION TO A VALUE

A related, and perhaps even more common, problem is to find what instruction is setting an improper value into some location. This can be accomplished with a variant of the break-time debugger "program" presented above.

```
#Break (at) 012345esc (after doing) Write esc
 ##If @012345 <> 067esc <then> ###Co (to) cr
 ##Mode Instruction esc ###Type @OLDCCcr
 ##cr
<Program number is [2]>  #
```

The important things to note about this example are the use of a conditional debugger command (if) and the "contents-of" operator (@). The first breakpoint subcommand states, in essence, "if the contents of location 012345 are not equal to the value 067 then continue execution; " the second subcommand states, "otherwise display the last instruction executed and break." The symbol-pair "<>" represents "not equal to" (literally, "less or greater"). When the program breaks, the offending instruction will have been discovered.

DETERMINE HOW MANY TIMES A CODE SEQUENCE IS ENTERED

Occasionally the efficiency of a program system is degraded by unnecessary and unexpected calls on subroutines that do initialization or other operations whose repitition do not cause errors but do affect performance. The PRIM debugger can be used to count the number of times a code sequence is entered. In the following example the code sequence is assumed to start at location 01234, and location 076000 (which is assumed to be unused) is used as a counter.

```
#Clear 076000cr
#Break (at) 01234esc (after doing) Xecute esc
 ##Set 076000esc = @076000+1cr
 ##Type 076000cr
 ##Go (to) cr
 ##cr
<Program number is [3]>  #
```

Every time location 01234 is entered, the count will be incremented and displayed. If only a final count is desired, rather than a running count of each execution, the following command could be used:

```
#Clear 076000cr
#Break (at) 01234esc (after doing) Xecute esc
 ##Set 076000esc = 076000+1cr
 ##Go (to) cr
 ##cr
<Program number is [4]>  #
 ...
#Type 076000cr
```

COUNT REFERENCES PRIOR TO A DESIGNATED CONDITION

The efficiency of a process can sometimes be evaluated by the number of times a data location is referenced prior to the occurrence of a given condition of interest. A variant of the previous example can be used where the automatic continuation is conditional on the designated condition not yet having occurred.

```
#Clear 076000cr
#Break (at) 012345esc (after doing) Read Write esc
 ##Set 076000esc = @076000+1cr
 ##If @5432 <> 0esc <then>  ###Go (to) cr
 ##cr
<Program number is [5]>  #
 ...
#Type 076000cr
```

When the condition that location 05432 contains a zero occurs, the program execution will break and the counter can be examined.

TRACE A LOOP ONLY ONCE

With the PRIM debugger, a program trace is accomplished by setting an execute break on all instructions of interest and supplying a break-time debugger program that displays the most recent instruction to execute (see the detailed example on entering and debugging a small program). To trace the loop only once, the automatic continuation is made conditional on CC not being equal to the starting location of the trace.

```
#Break (at) 01234:02345esc (after doing) Xecute esc
 ##Mode Instruction esc ###Type @OLDCCcr
 ##If @CC <> 01234esc <then>  ###Go (to) cr
 ##cr
<Program number is [6]>  #
```

To trace the loop n times, the continue could be made conditional on a counter that is incremented whenever the starting location is reentered. If a continuous trace is desired, the break can be entered for the event *STEP* and the go-to command can be unconditional; a ↑X abort intervention would be required to stop such a trace.

DETERMINE WHICH INSTRUCTIONS WERE *NOT* EXECUTED

An unusual use of the PRIM debugger is to determine what instructions were never executed while running some program. This can be accomplished quite easily by setting breakpoints throughout the area of interest and then having each execute-break remove its own breakpoint.

```
#Debreak (from) esc all (confirm)cr
#Break (at) 01234:05670esc (after doing) Xecute esc
 ##Debreak (from) @OLDCC:OLDCC+4cr
 ##Co (to) cr
 ##cr
<Program number is (7)>  #Break (at) 05675esc (after doing) Xecute cr
#Co (to) 01234cr
...
--> Break after executing 05675: J 010000  #Break (at) cr
03456-04557 <X>(7)  05675 <X>  #
```

At the end of the program, a display of the breakpoint data base shows those locations that were never executed. Note that when an execute breakpoint is specified over a range of locations it is detected only for the location containing the opcode portion of the instruction, so on each execution it is necessary to remove the breakpoints for all five instruction locations. Note also on the sixth line that even though all existing breakpoints had been deleted by the command on the first line, the new break-time program program was assigned the next higher sequence number (in fact, break-time programs are never deleted, so that they may be reused at a later time without having to reenter them).

DETERMINE WHEN DATA CHANGE OVER A CODE SEQUENCE

It is occasionally necessary to determine whether a code sequence has changed the value in some location. This can be done by setting a breakpoint at the beginning of the sequence, where the break-time commands copy the data value into an unused location, and setting another breakpoint at the end of the sequence, where the break-time commands compare the copied value with the current value. In the following example, the code sequence starts at 012345 and ends at 023456, the critical location is at 05432, and 076000 (assumed to be unused here) is used for temporary storage.

```
#Break (at) 012345esc (after doing) Xecute esc
 ##Set 076000esc = @05432cr
 ##Co (to) cr
 ##cr
<Program number is (8)>  #Break (at) 023456esc (after doing) Xecute esc
 ##If @05432=@076000esc <then> ###Co (to) cr
 ##cr
#cr
<Program number is (9)>  #
```

Each time the value in 05432 changes over the designate code sequence, program execution will break.

SEARCH A BUFFER FOR A GIVEN VALUE

The final example will show how to search a buffer (or any arbitrary set of locations) for the occurrence (or nonoccurrence) of a designated value. In this example, all values other than 3 in the left half of a word are to be found.

#Locate NON 0^esc (with mask) 070^esc (in) 013500:+3^cr
#

The general form of the locate command requires a comparison value, a mask, and a set of addresses to examine. The comparison value and mask can each be any arbitrary expression. The set of addresses can be a list of discrete locations or address ranges. By entering *NON* before the comparison value, its nonoccurrence is required. The comparison value defaults to "NON 0" and the mask defaults to "NOT 0" (i.e., all 1 bits). The test is performed by comparing the contents of each designated location with the aligned comparison value *only for those bits where the aligned mask bit is a one.* If all such bits compare, then a match has occurred; if any such bit differs, then a failure has occurred. The location in question is then displayed or not according to whether a match or non-match was specified. The locate command is analogous to the type command in that if it is terminated by an *escape* a replacement value can be entered for each displayed location. The same rules for the replacement value apply for locate-with-replacement as for type-with-replacement.

APPENDIX A: U1050 FEATURES

Memory Addresses:

$	Address of last cell displayed in the form "*address: contents*".
M*n*	Five-byte cell starting at address *n*, where *n* is assumed octal.
T*n*	Tetrad *n*, where *n* is octal or decimal.
IR.*n*	Index-register *n*.

Pushbuttons:

S.CLEAR	System-clear pushbutton.
T.LOAD	Load-from-tape pushbutton.
C.LOAD	Load-from-card-reader pushbutton.
GO.M	Next-instruction-from-M pushbutton.
GO.CC	Next-instruction-from-CC pushbutton.
OP.REQ	Operator-request pushbutton.

Control Counters:

OLDCC	Register containing address of last instruction executed.
CC	Register containing address of next instruction to be executed.

Indicators:

IND*n*	Testable-indicator *n*, where *n* is octal with no leading zero.
KNO	IND40.
KHI	IND41.
KEQ	IND42.
KUQ	IND43.
KLO	IND44.
AZ	IND45.
AN	IND46.
NBOF	IND47.
DOF	IND50.
IOSI	IND53.
NW2	IND54.
IOS3	IND55.
KBINT	IND56.
IRP or IOS2	IND57.
INH1	IND61.
SS1	IND62.
SS2	IND63.
SS3	IND64.
SI1	IND65.
SI2	IND66.
SI3	IND67.
TI	IND71.
NW	IND72.
CHINDS.*n*	I/O-indicator-bits for channel *n*.
IR1	Class-1-interrupt-inhibit flop.
IR2	Class-2-interrupt-inhibit flop.
IR3	Class-3-interrupt-inhibit flop.

Miscellaneous:

DAY.CLOCK	Day-time-clock (6 bytes).

Operators:

XOR	Exclusive-OR binary operator.
OR	Inclusive-OR binary operator.
AND	Logical-AND binary operator.
NOT	Logical-NOT unary operator.
<>	Not-equal-to binary relationshiop.
<=	Less-than-or-equal-to binary relationship.
>=	Greater-than-or-equal-to binary relationship.
<	Less-than binary relationship.
>	Greater-than binary relationship.
=	Equal-to binary relationship.
+	Plus binary operator or Increment unary operator.
-	Minus binary operator or Negation unary operator.
MOD	Modulus (integer remainder on division) binary operator.
*	Multiplied-by binary operator.
/	Divided-by binary operator.
ABS	Absolute unary operator (N.B. U1050 arithmetic is unsigned).
(	Left-bracketer.
)	Right-bracketer.
@	Contents-of unary operator.

Constants:

Octal	Digits 0-7, mandatory leading zero
Decimal	Digits 0-9, no leading zero
XS3	Prime-text-prime (XS3 characters converted to binary and truncated)
String	Quote-delimiter-text-delimiter (as in multi-word string "/text/)

Break Events:

ODD	Break-on-every-anomaly debug-switch.
PUT	Break-on-every-store debug-switch.
JUMPS	Break-on-every-jump debug-switch.
DROP	Break-on-every-conditional-jump-that-fails debug-switch.
STEP	Break-on-every-instruction-execution debug-switch
IO	Break-on-every-I/O-activity debug-switch.
INT	Break-on-every-interrupt debug-switch.
TICK	Break-on-every-clock-tick debug-switch.

Optional Anomaly Conditions (requested by ODD event):

Card Punch Feed-read
Card Reader 72-column Select
Device Not Ready
Disk Anomaly (bad function bits)
EOF Encountered in Communications Input
I/O Overlap
I/O Overload
I/O to Low Memory
Memory Exceeded
Nonexistent Controller

Printer 132-column Select
Tape Anomaly (bad function bits)
Too Many Product Bytes

Automatic Event Stops:
Halt-display Instruction
Hung on Class-1 Interrupt
Hung on Divide
Internal (emulator) I/O Synchronization Error
Program Halt
One-shot Error (no room for installation)
Unimplemented Instruction

Available Devices:

Printer:	XS3 translated to ASCII
	Speed parameter, defaults to 922 lines/ min.
Reader	12-bit binary, or ASCII translated to IBM026 column-binary
	Speed parameter, defaults to 600 cards/min.
Punch	12-bit binary, or IBM026 column-binary translated to ASCII
	Speed parameter, defaults to 300 cards/min.
Disk	Binary
Tape	Binary (or physical mag tape unit)
	Speed parameter, defaults to 3 min./2400-foot-rewind
LS-comm.	USAF6 translated to/from ASCII
	Speed parameter, defaults to 10 char./sec.
HS-comm.	ASCII (with bits 7 and 8 inverted) translated to/from ASCII
	Speed parameter, defaults to 240 char./sec.
Clock	XS3
	Rate parameter, defaults to 1 tick/sec.
Memory	Size parameter, defaults to eight 4192-word modules
	Cycle-time parameter, defaults to 4500 nanoseconds

APPENDIX B: U1050 CONFIGURATION PARAMETERS

Memory User settable speed parameter, defaults to 4500 nanosecond (nsec) cycle-time; can be set with a precision of 50 nsec. User settable size parameter, defaults to eight 4192-byte modules.

CPU Instruction timing is based on Univac Manual UP-3912 (Revision 1), Section 3, page 73. Timing is implemented as follows: (1) instruction fetches occur at memory speed; (2) indexing takes three additional memory cycles; (3) memory reads/writes by an instruction occur at memory speed; (4) any remaining time is divided by 4.5 microseconds (usec) to convert it to a corresponding number of memory cycles (and cause it to speed up or slow down proportional to memory speed).

Printer User settable parameter at installation time, defaults to 922 lines per minute.

Reader User settable parameter at installation time, defaults to 600 cards per minute.

Punch User settable parameter at installation time, defaults to 300 cards per minute.

LS-comm User settable parameter at installation time, defaults to 10 characters per second. Remote echo delay of one half a character time is provided. Up to 16 comm units (LS or HS in any mix) can be installed.

HS-comm User settable parameter at installation time, defaults to 240 characters per second. Remote acknowledgments occur after three character times. Up to 16 comm units (LS or HS in any mix) can be installed.

Disk Characteristics cannot be changed parametrically. Maximum disk address is 01453777. Disk timing is based on the following model: (1) one cylinder contains 64 sectors for 16 heads; (2) sectors of 168 bytes are interlaced on the disk in two cycles such that successively addressed sectors are separated by one sector from the other interlace cycle; (3) sector 0 on a cylinder starts just past the index point, and sector 32 arrives 6.267 milliseconds (msec) later; (4) one disk revolution takes 25 msec; (5) head positioning takes 10 msec for the first cylinder crossed and 111 usec for each successive cylinder; (6) transfer rate is 4 usec per byte. All timing is computed to a precision of 50 nsec.

Tape User-settable parameter at installation time, defaults to 3 minutes to rewind a 2400-foot reel. Other timing characteristics cannot be changed parametrically. Tape timing is based on the following model: (1) all tape records are one of two sizes, either 256 bytes or 3072 bytes; (2) reads or writes from load point take an extra 85 msec; (3) start-up of a read or write takes 10 msec if the last order completed more than 41 usec ago; (4) large block reads or writes take 129.396 msec, small blocks take 10.783 msec; (5) deceleration from a read or write takes 5 msec; (6) large block backspacing takes 144.496 msec, small blocks take 25.833 msec, but only the size of the last block read or written is remembered for timing purposes.

PRIM SYSTEM: USER REFERENCE MANUAL

INTRODUCTION

This document is the common reference manual for all users of the PRIM system, both those using one of the existing emulation tools and those writing new emulators. For the former, this manual is supplemented by the appropriate tool-specific guide (*e.g., PRIM System: U1050 User Guide*); for the emulator writer, the supplement is *PRIM System: Tool Builder Manual*.

The PRIM system is always in one of three states, known as the exec, the debugger, and the target execution states. The transition between states is controlled by the user. Both of the first two states are PRIM command processors that take commands from the user and execute them. The exec, whose command prompt character is ">", is used principally for setting up a target environment; the debugger, whose command prompt is "*", is used for the detailed examination and control of the executing target machine. Target execution includes the emulation of not only the CPU, but also clocks and assorted peripheral IO devices. The three sections following the introduction describe each of the states in turn.

The PRIM exec and debugger commands are illustrated with examples taken from *actual session transcripts. In all the examples, user input is italicized* to distinguish it from PRIM output. Input control characters appear as their abbreviations superscripted (*e.g.*, ^{esc}).

GENERAL INPUT CONVENTIONS

User input to PRIM, both exec and debugger, is generally free-format and case-independent. Leading spaces and tabs are ignored, and lower case is treated as its upper case equivalent (except in quoted strings, where case is potentially significant). User input to the target machine during target execution state is in the format required by the target system.

Certain characters have been assigned editing and intervention functions when input by the user. The editing characters apply only to the PRIM exec and debugger, while the intervention characters apply to the target execution state as well. The specific characters assigned to most of the functions may be altered (via the exec Change command) to suit one's needs. The editing functions are valid at any time during PRIM command input; commands are not executed until after the final character has been accepted.

Back-space (cntl-H) erases a character from the current word or term of input. The back-space is echoed as a backslash (\) followed by the erased character. When there are no erasable characters, a bell (cntl-G) is echoed instead.

Alternate back-space (initially cntl-A) performs a function identical to ***back-space***; it is provided as a convenience.

Backup (initially cntl-W) erases the current word or term of input. It is echoed as backslash (\) followed by the first character of the erased word.

Retype (initially cntl-R) retypes the current input line; it is useful after a confusing amount of editing has occurred.

Delete (initially DEL or RUBOUT) aborts the current input command or subcommand, allowing the user to re-enter it. It is echoed as " XXX".

Question (?), when entered at the beginning of a command field, elicits a description of the expected input, followed by a retype of the line. When the expected input is a selection from a list (or menu), the entire list is shown.

The intervention characters are valid at any time, including command input, command interpretation, and target execution.

Abort (initially cntl-X) interrupts the current activity and returns control to the command level of either exec or debugger. When used to cancel an exec or debugger command, control returns to the top level of the same state; abort is the only means of canceling a command when the user is in subcommand mode. When used to interrupt target execution, control returns to the state from which execution was initiated; abort is the only means of stopping a looping target machine.

Status (initially cntl-S) produces a one-line summary of target machine status, including program counter, emulated elapsed time, and active IO devices. The command is valid at any time, but useful primarily in execution state.

The following character is active only during target execution.

Control-shift (initially cntl-1) permits the user to enter (during execution) a control code that cannot be entered directly because it is intercepted by either PRIM or the operating system; the PRIM characters inolved are ***status***, ***abort***, and ***control-shift*** itself. The next ASCII character following the ***control-shift*** (other than the digits 0 thru 9) has its two leading bits cleared, thus converting it to an ASCII control code (*A* or *a* to *cntl-A*, etc.). ***Control-shift*** followed by a digit results in an input that is outside the normal target character set and is used for particular target-machine-dependent functions. The control-shift character itself is not echoed, and not passed to the target machine. If execution terminates before that next character is input to the target device, the control-shift is canceled; it is not retained for the next resumption of execution.

PRIM EXEC

The PRIM exec is the initial state of a PRIM session. Exec commands are concerned primarily with building target configurations, saving PRIM session results, restoring previously saved sessions, and accessing or creating files (within the file space of the host operating system).

The exec prompt character is ">", indicating that PRIM is in exec state and that the exec is awaiting a new command; it is always shown on a new line. Individual input fields consist of keywords (a word selected from a menu), decimal numbers, and file names. Exec commands are composed of fixed sequences of fields, each terminated by a delimiter character; a final confirmation consisting of a *return* is often required.

Keywords are selected by any unambiguous leading substring. Often, a single character suffices; three characters are always sufficient. Numbers are specified in their entirety. File names are specified according to the conventions of the operating system. All commands that will use a file for output require the name of a new file (except the Mount-Append and Mount-Old commands, which modify existing files); all other file commands require the name of an existing file. In TENEX, an existing file name -- and a new file that is a new version of an existing file name -- is recognized (and completed) in response to an input *escape*.

The normal delimiters that terminate command fields are *return*, *escape*, and *space*. *Escape* and *space* function identically except that the former generates feedback to the user while the latter generates none; the feedback produced by *escape* includes both field completion and next-field prompting (which is given in parentheses). *Return* is used to complete a command immediately, bypassing any remaining fields and confirmation; if further input is required, the *return* is treated as an *escape*. (In the examples that follow, *escape* termination is used to show the prompts.)

Keywords that involve either devices or parameters are machine-dependent; the selections shown in the examples are meant to be illustrative rather than definitive. Device specification is further complicated when two (or more) of the same generic device are installed. Therefore, for device names, two further delimiters are utilized, *at* ("@") and *colon* (":"). A fully qualified device name consists of *generic-name* @ *channel-number* : *unit-number*; the numbers are required only to the extent necessary to specify a particular device. When a device name is terminated by one of the standard terminators, and when further disambiguation is required, the exec prompts explicitly regardless of the terminator.

The remainder of this section consists of the descriptions of the exec commands in alphabetical order. Each command description begins with a transcript showing one or more examples of the command and its various options. Those commands that require a second keyword show that list via an input *question*. The exec commands are:

```
>? One of the following:
 CANCEL
 CHANGE
 CLOSE
 COMMANDS
 DEBUG
 FILESTATUS
 GO
 INSTALL
 MOUNT
 NEWS
 PERIPHERALS
 QUIT
 REASSIGN
 RESTORE
 REWIND
 SAVE
 SET
 SHOW
 SYMBOLS
 TIME
 TRANSCRIPT
 UNINSTALL
 UNMOUNT
>
```

Comment.

```
>; this line is a comment<cr>
>
```

Any line beginning with a *semicolon* is treated as a comment. Comments are recorded in the transcript if one is open (see Transcript command).

Cancel abandons all outstanding IO operations for a designated device.

```
>ca<ESC>NCEL (IO for device) ta<ESC>PE-UNIT <cr>
>
```

This command is intended for use when, after an IO error halt (described in the section on target execution), the user wishes to abandon the device operation rather than mount a file and retry the operation. The list of outstanding IO operations, by device, is part of the Peripherals command output.

Change reassigns the PRIM control functions.

```
>ch[esc]ANGE (input code for) ? One of the following:
 ABORT
 ALT-BACKSPACE
 BACKUP
 DELETE
 RETYPE
 STATUS
 CONTROL-SHIFT
>CHANGE (input code for) ab[esc]ORT (from ↑X to) ? A Control Code.
>CHANGE (input code for) ABORT (from ↑X to) ↑P cr

>ch[esc]ANGE (input code for) d[esc]ELETE (from <DEL> to) [esc] (not changed)
>
```

This command allows the user to change the ASCII control code assigned to any of the listed PRIM control functions from its current assignment to another (currently unassigned) control character. The function name is the second word of the command; when it is terminated with an *escape*, the current assignment is noted in the noise. The entire set of ASCII control codes (including *delete*) is available excepting *null*, *back-space*, *line-feed*, *return*, *escape*, and *unit-separator* (TENEX *end-of-line*) which have fixed functions in PRIM. For *abort* and *status* the set is limited to *cntl-A* thru *cntl-Z*.

Close terminates the current transcript file if one is open.

```
>cl[esc]OSE (transcript file.) cr
>
```

A transcript file is opened using the Transcript command; it is automatically closed at the end of a session.

Commands redirects subsequent input from a file.

```
>co[esc]MMANDS (from file) command.file[esc] cr
>
```

This command causes PRIM to read its subsequent command input from the named file instead of the user terminal (or current command file). The file input is treated exactly as terminal input except that intervention functions (*abort* and *status*) are valid only from the terminal. Should a command in the file cause execution to be resumed, input that normally would come from the user terminal is taken instead from the file. Input reverts to the previous source at the end of the file; an *abort* terminates all command files and reverts input to the user terminal. Command files may be nested. Command files are very useful for common session-initialization sequences.

Debug transfers control to the PRIM debugger.

```
>d[esc]EBUG
#return (to EXEC) cr
>
```

The PRIM debugger is described in the next section; control is returned to the exec via the debug Return command.

Filestatus returns information about mounted files for all or designated devices.

```
>FILESTATUS (for device) ESC ALL
 Record   File Name          Device
 12       CARD.DECK          CARD-READER
 12       User Tty           PRINTER
 825      TERMINAL.INPUT     TERMINAL (In)
 12345    TERM.OUT           TERMINAL (Out)
 2+6      ARCD.EFG           TAPE-UNIT:0

>FILESTATUS (for device) CARD-READER
 Record   Type  Byte/Last     File Name
 12       Bin12 960/1280     CARD.DECK
>
```

When the device field is empty (*return* or *escape*) all mounted files are listed; otherwise just the file(s) on the named device are listed. The latter case gives more complete status than does the former. The output fields are:

Record tells the current position of the device or the number of records which have been processed. For disks, it is a sector number; for card readers and punches, a card count; for communication lines, the total number of bytes transferred; for mag tape units, the position from beginning of tape expressed as files + records.

File Name is the name of the file; the name "User Tty" is displayed when *THIS-TERMINAL.* is the file.

Device is the emulated device on which the file is mounted.

Type describes the type of file, either Ascii or Binxx, where xx is the file byte size. The type may have been explicitly specified at mount time, or it may have been assumed by PRIM.

Byte/Last is, for a mounted disk file, the current byte position in the file and the total number of bytes in the file.

The marginal notation "[not opened]" indicates that the named file could not be found (this occurs only to a restored file) and that the device must be reassigned to another file (or to the same file via a new path name).

Go transfers control to the target execution state.

```
>GO (from 1234) CR
--> MACHINE running at 5670, Used 0:00.4
--> MACHINE halted at 6543, Used 0:01.0
>
```

This command transfers control from the PRIM exec to the emulator or target machine, in its current state. Control returns to the exec when the target machine halts or a breakpoint is encountered (see the debugger Break command) or the user interrupts execution with an *abort*.

In the example, the user followed the command with a *status* request (the *status* character itself is not echoed) resulting in the first reply line (MACHINE running at ...); the target machine is still running. Eventually the target machine halted, producing the second status line and returning control to the exec as evidenced by the exec prompt.

Install adds a designated type of device to the machine configuration.

```
>i^escNSTALL (device) ? One of the following:
 CARD-READER
 PRINTER
 TAPE-CONTROLLER
 TERMINAL
>INSTALL (device) p^escRINTER (CHANNEL) 1^esc
>>? SPEED
>>s^escPEED (characters per second) ^esc300
>>^cr

>i^escNSTALL (device) ta^escPE-CONTROLLER (CHANNEL) 3^esc cr
How many TAPE-UNIT's do you want? 2^cr
For the first TAPE-UNIT, (UNIT) 0^esc cr
>>^cr
For the second TAPE-UNIT, (UNIT) 1^cr
>>^cr
>
```

The device type is selected from among those implemented. The user is prompted for each necessary item of information, typically including an address for the device in the target IO address space and the number of units to install. After the required information is gathered, sub-command mode (">>" prompt) is entered to gather optional parameters; any optional parameter not supplied takes on its default value. Subcommands are terminated by an empty command, *return* only. *An installed device is initially unmounted* -- there is no file associated with the device for purposes of actual IO.

When the device being installed is a multi-unit controller, the dialogue proceeds through each of the individual units to gather their parameters. After the command is completed, the controller is no longer visible; only the individual units are. An *abort* aborts the entire command, not just the current unit.

Installation is permitted only before any execution has taken place. Typically, a user or user group installs a standard configuration and then saves it for use in all subsequent sessions (see the Save-Configuration and Restore commands). The optional parameters of an installed device may be changed at any time using the Set command.

Mount associates a file with an installed device.

```
>mASCOUNT (A,I,N,OL,OU,T,?) ? One of the following:
 APPEND
 INPUT
 NEW
 OLD
 OUTPUT
 THIS-TERMINAL
>MOUNT (A,I,N,OL,OU,T,?) tASCHIS-TERMINAL  (on device) pASCRINTER cr

>mASCOUNT (A,I,N,OL,OU,T,?) nASCEW (in & out file) ABCD.EFG;1ASC (on device)
 taASCPE-UNIT cr
>

>m iASCNPUT (from file) card.deckASC (on device) caASCRD-READER cr
>>? BINARY or ASCII
>>bASCINARY (with byte size) 12cr
>>cr
>
```

Associating a file with an installed device causes subsequent emulated IO for that device to be directed to the file. The second keyword following Mount determines the direction of data flow and the choice of an old (existing) or new file. A file must be mounted on a device before any actual IO can take place.

APPEND mounts an old file for output only, with the subsequent output being appended to the previous contents of the file.

INPUT mounts an old file for input only.

NEW mounts a new file for both input and output (the file is initially empty).

OLD mounts an old file for both input and output (subsequent output overwrites any existing file data).

OUT mounts a new file for output only. For a disk or tape device, OUT is treated as NEW.

THIS-TERMINAL associates the user terminal -- instead of a named file -- with the named device. The mounting is for both input and output unless a file has already been mounted for one, in which case the terminal is mounted only for the other. The terminal is known to be an ASCII "file". The terminal may be mounted only once for input; it may be mounted for output (or on an output-only device) any number of times, but the output is not labeled as to source.

Only some of the forms above are applicable to any given device. For a disk- or tape-like device, an INPUT, OLD, or NEW file is expected; an OLD file is one that was NEW in a previous PRIM session, and is being re-used, while an INPUT file is an old read-only file. For a bidirectional communication device (*e.g.*, a terminal), two files are required: an INPUT file and either an OUTPUT or APPEND file. Alternatively, a real terminal may be used for both (or either one). For an input-only device, INPUT and OLD are identical; for an output-only device, OUT and NEW are identical.

For those devices that deal exclusively with character data, the mounted file is always taken as an ASCII text file; character translation is performed as part of the IO process. (This allows the file to be created and/or processed by any operating system utility that deals with text files.) For tape and disk devices, the file format is internal to PRIM (and therefore not requested from the user); the data is recorded directly. For other devices the user is asked, via subcommand mode (">>" prompt), whether the mounted file (NOT the device) is an ASCII text file or a binary file containing a stream of pure data in bytes of some fixed size. The default is a binary file of a device-dependent byte size.

Once a file has been mounted on a device, all exec commands that refer to the file require the device name as the specifier; for communication devices, where two files are normally mounted, the device name is followed by a direction selector. The file name itself is not used as the internal identifier.

News reads the PRIM on-line news file.

```
>n^ASCEWS
 Do you want to see  4-APR-77  Changes in PRIM ?: ^ASCYES
  [  Here comes the message regarding changes of 4-APR-77 ...  ]
 Do you want to see 24-MAR-77  Preliminary Documentation ?: del XXX
>
```

The date of the most recent news message is shown automatically at the start of each session. In response to the command, each message's date and subject is shown, beginning with the most recent message. For each message, the body may be seen (*YES*) or skipped (*NO*), or the command may be terminated (*delete* or *abort*).

Peripherals returns information about the installed devices.

```
>p^ASCERIPHERALS
Chan  Unit  Mounted  Device
 1     0      No     PRINTER
 2     0      Yes    TERMINAL
 3     0      Yes    TAPE-UNIT
 3     1      Yes    TAPE-UNIT

 active devices: TERMINAL
>
```

This command produces a listing of all the installed devices, together with their IO addresses and a notation concerning whether they have files mounted. It also lists all devices which have suspended IO operations. Ordinarily, suspended operations are limited to (1) IO error conditions and (2) input operations where the input file is a real terminal and no input was available when target execution stopped.

Quit terminates a PRIM session.

```
>q^ASCUIT
  Quitting MACHINE  (Confirm) cr
e
```

Terminating the PRIM session involves closing all open files and returning control to the process that initiated the PRIM session. The session cannot be continued.

Reassign specifies a new file for a mounted device.

```
>reassign (device) in-type-unit (to file) new_file cr
>
```

This command is used to substitute a new file specification when, after a prior Restore command, a previously mounted file cannot be found. In particular, a restore done from a different directory than the one in force at save time has trouble finding any of the mounted files. Reassign may only be used for devices/files that are marked "[not opened]" in a file-status display. The new file is assumed to have the same characteristics as the old one and is positioned at the same file position.

Restore recovers the state information saved in a file.

```
>restore (from SAVE file) >UDD>CONFIG;1 cr
  restored CONFIGURATION from TUESDAY, MAY 3, 1977 12:35:08 PDT
>
```

The current context is updated with the complete or partial environment previously saved in the designated file by the Save command. For the addressable regions -- machine memory, registers, etc. -- the saved data replaces the current data only for those cells that were actually saved; cells not saved are not cleared. (Thus, nonoverlapping memory images are merged.) For nonaddressable regions -- symbol, configuration, and breakpoint -- each one is completely replaced if present in the file. The date and region(s) saved are shown, followed by a list of any mounted files that cannot be found.

Rewind returns a device's mounted file(s) to the beginning.

```
>rewind (device) in-type-unit cr

>rewind terminal (B,I,O,?) ? One of the following:
 BOTH
 INPUT
 OUTPUT
>REW TERMINAL INPUT cr
>
```

This command is useful for retrying a program without unmounting and remounting files. (Files are always rewound when mounted, except for Append files, which cannot be rewound.) For a terminal-like device that requires separate input and output files, the user optionally specifies which file is to be rewound; the default is *BOTH*.

Save copies selected state information into a file.

```
>save ? One of the following:
 ALL
 CONFIGURATION
 FORMATS
 MEMORY
 SYMBOLS
>SAVE configuration (on file) >UDD>CONFIG;1 cr
>
```

This command saves on the (new) file an image of the region(s) selected for saving. The contents of the file can later be restored for use in this or another session. The second word of the command selects one of the save options.

ALL saves everything -- a complete checkpoint of the target machine and debugging state. "Everything" includes memory, all addressable registers, installed devices, mounted files together with their positions, debug breakpoints and their programs, debug formats and modes, defined symbols, and the internal state of the emulated machine.

CONFIGURATION saves all the machine configuration data, including installed devices, mounted files (if any), machine parameters, and debug formats and modes. This command is allowed only before any execution takes place. Useful for creating a standard machine configuration (possibly with some standard files mounted) for use in subsequent sessions.

FORMATS saves all the formats that have been defined (using the debugger Format command).

MEMORY saves those regions of the machine memory that are not clear. (At the start of a PRIM session, memory is already cleared.)

SYMBOLS saves all the user-defined symbols, both those loaded via the exec Symbols command and those defined directly via the debugger New-symbols command. The file that results is a SAVE/RESTORE file, not a SYMBOLS file!

Set changes the values of user-settable parameters.

```
>set SET (<empty> or device) cr
>>? One of the following:
 CLOCK
 MEMORY
 SPEED
>>CLOCK (ticks per second) 1000 cr
>>MEMORY (8K modules) 4cr
>>cr

>set SET (<empty> or device) PRINTER
>>SPEED (characters per second) 150cr
>>cr
>
```

Following the command word, the user selects the group of parameters he wishes to alter. An immediate *return* selects the global machine parameters; a device name selects the parameters of that particular installed device (the parameters of multiple installed instances of the same device type need not have identical settings).

Any number of parameters from the selected group may be changed. In response to the subcommand prompt (">>"), the name of a parameter and its new value are entered; each change is made immediately and a new subcommand prompt appears. The command is terminated by an empty input, *return* only, or by an *abort* (which does not undo any parameters previously changed). The list of possible parameters is highly machine- and device-dependent; it typically includes the size of memory and the speed of each device.

The value of a parameter is either a (decimal) number or a keyword from a parameter-specific list; a *question* in the value field reveals which is expected. An *escape* sets the parameter to its default value.

Show displays the values of all the parameters in a group.

```
>show (<empty> or device) cr
 CLOCK is 1000 ticks per second
 MEMORY is 4 8K modules
 SPEED is 750 nanoseconds per memory cycle

>show (<empty> or device) printer
 SPEED is 200 characters per second
>
```

Following the command word, the user selects either the global machine parameters (*return*) or the parameters of an installed device. The names and current values of all the parameters are displayed.

Symbols reads an ASCII symbol-table file.

```
>symbols (from file) SYMBOLS.EXAMPLE.asc cr
>
```

This command causes PRIM to build a user-defined symbol table from the data in the named file, which is a structured ASCII text file. The file may define values for both global symbols and program-local symbols that are organized into programs. In the PRIM debugger, the global symbols plus the local symbols of the currently open program are accessible at any time. Symbol values in the file are octal. The form "name == value" defines a global symbol; the form "name = value" defines a local symbol; the form "name:" establishes a program name to which subsequent local symbols are assigned. The file is free-format in that spaces, tabs, commas, and new-lines may occur anywhere -- except in the middle of names or values. The following is a sample symbols file.

```
ALPHA==45
BETA==12345
PR1: A=2000, B=2132, C = 2241
XYZ:
A=3212  AA=3245, AAA=3261,AAAA=7777
```

Symbol files are intended to support the moving of symbolic label data from an assembler or linking loader into PRIM for use in symbolic debugging.

Time displays time-of-day and time-used information.

```
>time (is) TUESDAY, MAY 3, 1977 12:34:33-PDT
 Used 0:14.6 PRIM time;  Used 0:02.7 MLP time.
>
```

This command displays the date, time of day, the amount of PRIM time used and the amount of MLP-900 time used in this PRIM session. (Elapsed target machine time is displayed in response to *status*.)

Transcript transcribes the subsequent PRIM session on a new file.

```
>transcript (to file) newfile.asc cr
>
```

All transactions with the user terminal, including execution-time IO to THIS-TERMINAL, is transcribed until either the user terminates the session (with a Quit command) or closes the transcript. Only one transcript may be open at a time. A header line containing the date and time is placed at the head of the file.

Uninstall removes an installed device.

```
>UNINSTALL (device) ? PRINTER or TAPE-UNIT
>UNINSTALL (device) TAPE-UNIT (unit): 1 esc cr
>
```

This command is the inverse of the Install command; it removes an installed device from the configuration, first unmounting its files if necessary.

Unmount unmounts the file(s) from a device.

```
>UNMOUNT (device) PRINTER cr

>UNM TERMINAL (B,I,O,?) ? One of the following:
 BOTH
 INPUT
 OUTPUT
>UNM TERMINAL esc BOTH cr
>
```

The unmounted file(s) are closed. For a terminal-like device that requires separate input and output files, the user optionally specifies which file is to be unmounted; the default is BOTH.

PRIM DEBUGGER

The PRIM debugger is a table-driven, target-machine-independent, interactive program for debugging a PRIM emulator or a target program running on such an emulator. It is tailored to a specific target machine by tables prepared as part of an emulation tool. Basically, it permits a user to set and clear breakpoints and to examine, modify, and monitor target system locations. Target system assembly language and symbolic names are recognized, and arithmetic is performed according to the conventions of the target machine. The debugger command prompt character is "*e*"; each level of subcommand adds another "*e*" to the prompt.

ARGUMENTS

Most debugger commands take arguments in the form of values, expressions, expression-ranges, lists of expressions, or lists of expression-ranges as defined below.

Values

A value is an assembly-language instruction, a form, text, or an expression-list. Assembly language instructions are parsed by a table-driven assembler/disassembler that accepts the same syntax as the assembler for the target machine. User symbols will be recognized if they have been supplied in user symbol-table files (see the exec Symbols command) or have been declared individually (see the debugger New-symbol command).

A form requires that the user previously define a corresponding format (see the debugger Format command). A form is represented by the format name followed by an expression-list, as in the following example.

F1 0, 7, 3

Text is represented as a double-quote ("), followed by an arbitrary delimiter character, followed by a sequence of other (non-delimiter) characters, followed by another occurrence of the delimiter character, as in the following example.

"/This is text./

Expressions

An expression is any well-formed sequence of constants and symbols that are defined for the target machine; the symbols (which are machine-specific) may represent either locations or operators whose rules of combination determine what is a well-formed expression. A location symbol may represent a named hardware element or a globally or locally defined user location. An operator may either be unary (preceding its operand) or binary (coming between its operands in infix notation). The precedence of operators is a function of the target machine, except that all unary operators are assumed to have the same precedence value, which is higher (more strongly binding) than that for any binary operator. If brackets are permitted (*e.g.*, parentheses), their precedence value is higher than that of unary operators. For example, A-B and -B+A will evaluate the same, but will differ from -(B+A), which will evaluate the same as -B-A. A bracketed subexpression may itself attain the full complexity of an expression. The behavior of operators is machine-specific.

Expression ranges

An expression-range consists of the triple: expression (lower bound), colon, expression (upper bound). It represents a sequence of locations starting at the lower bound and continuing through successive locations to include the upper bound. The upper bound may not be less than the lower bound. Wherever an expression-range is allowed, a single expression is accepted and treated as if it had been entered as both the lower and upper bounds of a range. If the two bounds in a range address different spaces (see the discussion of Spaces below) within the target machine, the sequence of locations is restricted to that space addressed by the lower bound. Two special forms of expression ranges are recognized. If the second expression in a range is "-1", it is treated as being the largest address in the space referenced by the first expression. If the second expression in a range is of the form "+ *expression*", it is treated as if it were "*(lower bound)* + *expression*."

Lists of expressions or ranges

A list of expressions consists of at least one expression, followed, optionally, by any number of occurrences of a comma followed by an expression. A list of expression-ranges has the corresponding structure of at least one range, followed, optionally, by any number of occurrences of a comma followed by a range. An example of a list of ranges is

0:10, 20, 30:50

Note that the second element of the list (20) is an example of a range with a defaulted upper bound.

SPACES

Addressable locations in a target system are organized into constructs called spaces. A space consists of a set of addressable locations that is closed under a successor function and its inverse (a predecessor function). For example, main memory constitutes a space, typically starting at location zero and continuing through an arbitrary number of locations. The successor to the last element of a space is the first element in that space; and the predecessor of the first element is the last one. In some cases, machine locations are grouped into a space for convenience, even when the concept of a successor function for elements of that space has no correspondence in the actual target system. Such a space might consist of testable indicators. The machine symbols are identified in the tool-specific user guide.

For purposes of the debugger, every addressable location in a target system is represented by a pair: (*space, element*). When a range is specified, two such pairs (*a,b*):(*c,d*) are implied. To avoid ambiguities where *a* and *c* differ, the debugger ignores *c* and treats such a range as a sequence of locations, all in space *a*, starting with element *b* and continuing through element *d*.

SYNTACTIC UNITS

The basic syntactic units the debugger deals with are

1. Literals
2. Symbols
3. Punctuation

Literals

Literals are character constants, numeric constants, or single characters that have some encoded meaning (which may be context-dependent). A character constant is supplied to the debugger as a machine-specific character-constant prefix string followed by a string of data characters of arbitrary length, followed by a machine-specific character-constant suffix string of the general form:

prefix-string character-data-string suffix-string.

If the first character of the suffix string is to be included in the data string, it must appear doubled. Character constants are converted to binary (right justified) and are truncated to fit the element in question. As the form of a character constant is machine-specific, it is described in the tool-specific user guide.

A numeric constant is supplied to the debugger as a machine-specific (and optional) radix-prefix string followed by a string of digit characters followed by a machine-specific (and optional) radix-suffix string of the general form:

prefix-string digit-string suffix-string

The prefix and suffix strings establish the radix within which the digit characters are evaluated. The digit characters for any radix *r* are the first *r* characters of the set {0,...,9,A,...,Z}.

Coded characters have independent meaning only within certain contexts: at appropriate points in the dialogue they designate a particular debugger command, a mode, a breakpoint type, etc.

Symbols

There are five types of symbols: machine symbols that are assigned to hardware elements in the target machine, predefined opcodes for symbolic instructions, user-supplied names of formats, operators for expressions, and user symbols that can be assigned to arbitrary memory locations. Machine symbols are given in the tool-specific user guide; other symbols are assumed to be familiar to the user.

User symbols are either loaded from a file using the exec Symbols command or individually defined using the debugger new-symbol command. The symbols include both global symbols and program-local symbols that belong to specific named programs. The global symbols are available at all times; the program-local ones only when theirs is the open local symbol table.

Punctuation

Punctuation marks are characters with a predefined syntactic (and usually semantic) role. The punctuation characters are the separators (*comma* and, in format definitions, *space*), the terminators (*return*, *escape*, and, in replacement operators, *back-slash* and *up-arrow*), and a semantics-free delimiter (*space*). *Escape* is used as a terminator instead of *return* to invoke a subcommand or an additional feature of a command (*e.g.*, in Mode or Breakpoint commands described below).

ERROR DETECTION AND EDITING

Debugger commands are examined for errors as they are entered, character by character. As soon as an error has been detected, a bell (beep) is echoed and further input is rejected, except for the generic editing characters *back-space*, *retype*, *backup*, *delete*, or *abort*.

COMMANDS

Debugger commands are all single characters; they can be organized into several groups: debugger control, execution control, display, and storage. Each is listed below. Unless otherwise indicated, the command character is the first character of the command name.

Debugger Control

Debugger Control commands provide for user control over several aspects of the behavior of the debugger. They permit the user to execute commands indirectly or conditionally, to return from the debugger to the PRIM exec, and to control the debugger's representation of data. The Debugger Control commands are:

Use. Calls a designated break-time program as if some breakpoint associated with that program had just occurred. A program number must be designated that corresponds to an existing break-time program. Program numbers are shown when the breakpoint data base is displayed (see the break command); the program itself can be seen using the program-edit command.

```
#Use-program ?(number of an existing break program)
#Use-program 2cr
```

If the use command is itself in a break-time program, then a go command executed in the called program causes termination of the calling program as well as of the called program.

If. Tests the supplied expression and, if it is true, executes the following subcommand. A true expression is one whose value is *odd*; relational operators yield a value of one when true and zero when false. The tested expression must be terminated by an *escape*.

```
#If ?(expression)
If 3esc <then> ##Type 0cr
 00: 00   #

#If 2esc <then> ##Type 0cr
#
```

Return. Returns control to the PRIM exec; confirmation is required.

```
#Return (to EXEC) cr
>
```

Mode. Interrogates default and current modes and changes modes. A *question* after the command character *M* will elicit the default and current mode setting; another *question* will list all mode settings and associated mode-code-characters.

```
#Mode ?
Current and (Default) mode settings:
  Feedback          Verbose         (Verbose)
  Output            Bits            (Bits)
  Addresses         Symbolic        (Symbolic)
  Line-format       Dense           (Dense)
  Radix             8               (8)
Type ? for more

#Mode ?
Feedback:
  C     Concise
  V     Verbose
Output:
  B     Bits
  F     Formatted (format-name)
  I     Instruction
  N     Numeric
  T     Text
Addresses:
  A     Absolute
  S     Symbolic
Line-format:
  D     Dense
  E     Expanded
Radix:
  Rn    Radix-base n (1 < n < 37 decimal)
#
```

A list of mode settings is expected following the Mode command; if none is supplied, the default settings are reestablished. If the list is terminated by a *return*, the current modes are changed. If the list is terminated by an *escape*, a temporary change is made that applies only to the following subcommand, as in the following example.

```
#Mode Instruction esc #Type 01234cr
 01234: JUMP 0567
#
```

Modes are established for feedback (verbose or concise); output (bits, formatted, instruction, numeric, or text); addresses (absolute or symbolic); output line format (dense or expanded); and output radix (any base from 2 through 36).

The feedback modes control how debugger commands are reflected to the user: *concise* suppresses all "noise" feedback (such as command completion); *verbose* enables it. The output modes control the general representation of data: *bits* treats a datum as an unsigned magnitude; *formatted* treats it as a pattern of bits partitioned into contiguous fields according to a designated format (see Format command); *instruction* treats it as a machine instruction and disassembles it; *numeric* treats it as a signed value, if that is appropriate for the machine; and *text* treats it as a representation of a string of characters. The address modes control whether numeric-mode values are to be converted to symbols (if possible): *absolute* suppresses the symbol look-up; *symbolic* enables it. The line-format modes control the density of displays: *dense* suppresses most

debugger-generated line-feeds so as to show more information per line, *expanded* enables them.

When formatted output is selected, the name of the output format must be specified, as in:

```
#Mode Formatted F1 cr.
```

Output radix sets the number base for the representation of numeric data (note that numeric input data self-identify the number base). For example,

```
#Mode Radix 16 cr
```

causes current output radix to become hexadecimal.

Format. Permits the user to name and define a format as a list of fields, each of which is a designated number of bits wide. The field widths are supplied as a list of numeric constants (separated by commas or spaces).

```
#Format F1esc 2 4 6 8cr
#

#Mode Formatted F1 esc #Type 0cr
 00: 00,00,00,00  #
```

If the format command is terminated without having defined a format, all defined formats are displayed, as in

```
#Format cr
 F1 2,4,6,8  #
```

Comment. Following an initial semicolon, ignores all subsequent inputs up to and including a line terminator.

```
#; THIS IS A COMMENT--IT DOES NOT GET INTERPRETED.cr
#
```

New-symbol. Adds a list of new user symbols to the (possibly empty) global symbol table. Each new symbol in the list is supplied as a name followed by a *space* or an *escape* followed by an expression giving its location.

```
#New-symbols ?(((new-symbol) <ESC> (expression))-list)
#New-symbols PATCHesc <at> 070000cr
#Type PATCH,PATCH-1,PATCH+1cr
 PATCH: 00    067777: 00    PATCH+01: 00  #
```

Kill-symbol. Removes a list of user symbols from the open local or global symbol table.

```
#Kill-symbols ?(list-of-user-symbols)
#Kill-symbols PATCHcr
#Type 067777:+2cr
 067777: 00    070000: 00    070001: 00  #
```

Open-symbol-table. Opens a local (program-specific) symbol table if one is specified; the currently open local symbol table, if any, is closed in any case. After this command is executed, the available symbols include the global symbols plus the local symbols of the specified program; if no program is specified, only the global symbols are available.

```
#Open-program-symbols ?(program-name) or not <close the open local symbol table>
#Open-program-symbols cr
#
```

Execution Control

Execution control commands provide for user control over execution of the target program. They permit the user to continue execution, transfer to a designated location, set and clear breakpoints or edit break-time programs, and single-step the target program. The execution control commands are

Go. Passes control to the target machine in its current state. If an argument is supplied, its value is first stored into the program counter. The argument can be an arbitrary expression, so long as it evaluates to a legal memory address.

```
#Go (to) ?(expression) or empty
#Go (to) 01000cr
```

Break. Displays or sets breakpoints in the target machine. The two classes of breakpoints are known as event breakpoints and reference breakpoints. There is a fixed set of event breakpoints defined for any given target machine; each describes a type of event whose occurrence causes the emulator to break if the corresponding event breakpoint is set. The set of event breakpoints always includes (1) every instruction-execution (single step), (2) every branch of control, and (3) every memory write; other events are defined for each machine as appropriate. Reference breakpoints cause the emulator to break when a specific type (read, write, and/or execute) of reference to a specific location occurs. Reference breakpoints may always be set on memory locations; other spaces in which reference breakpoints may be set are detailed in the tool-specific user guide. Any number of reference breakpoints may be set at any time.

The break command followed immediately by a *return* causes all existing breakpoints (*i.e.*, those in the breakpoint data base) to be displayed; if a break-time program is associated with a breakpoint, its number is also displayed. Otherwise, a list of either events or ranges (reference locations) for the setting of breakpoints is supplied. If a list of ranges has been entered and terminated with an *escape*, then a list of read, write, or execute reference-break conditions is specified next (as permitted at those locations); the default is all three types. Whenever a breakpoint is set for an event or a location, any earlier breakpoint for that same event or location is superseded.

If the list of events or break types is terminated by an *escape*, as in the second example below, a break-time "program" may be supplied to be executed by the debugger when the break is encountered. The following commands are permitted within such a break program: Clear, Comment, Debreak, Evaluate, Go, If, Jump-history, Locate, Mode, Open, Set, Type, and Use. Replacement within a locate or type command is not permitted in a break-time program. Any number of commands can be included in a break program; the program is terminated by an empty command (terminator only).

```
#Break (at) ?(event-list) or ((expression-range)-list) or <RETURN>
<? for list of events>
#Break (at) 0123:0456, 0712esc (after doing) cr
<R,W,X>  #

#Break (at) 01000esc (after doing) Xecute esc
 ##Type 0cr
 ##Go (to) cr
 ##cr
<Program number is [1]>  #

#Break (at) TICKcr
#
#Break (at) cr
0123-0456 <R,W,X>  0712 <R,W,X>  01000 <X>[1]  TICK <event>  #
```

During program execution, if an event break is detected, or if a reference break (read, write, or execute) is detected at a location for which the corresponding break type has been specified, then execution is terminated before beginning the next target machine cycle and control passes to the debugger to process the break. If a break-time program has been supplied for that break event or location, the program's commands are executed in order by the debugger until either a go command or the end of the program is encountered. If several breaks occur on the same cycle, the program associated with each of them is executed; the order of break-program execution corresponds to the order in which the breaks are reported by the emulator. If every break causes execution of a Go command, then the target program is automatically resumed, provided there is no ambiguity as to where execution is to resume. Otherwise (*i.e.*, if any break had no program or failed to execute a Go command), a message describing each of the breaks is displayed and the normal command level of the debugger is entered.

Debreak. Clears event breakpoints or reference breakpoints at locations in the target machine. The default is to clear all breakpoints. Examples of debreak commands are

```
#Debreak (from) 0234:+4cr
#Break (at) cr
0173-0233 <R,W,X>  0241-0456 <R,W,X>  0712 <R,W,X>  01000 <X>[1]
TICK <event>  #

#Debreak (from) esc all [confirm]cr
#Break (at) cr
#
```

Program-edit. Displays a designated break-time program or permits it to be edited. A program number must be designated that corresponds to an existing break-time program. Program numbers are shown when the breakpoint data base is displayed (see the break command). If the command is terminated by a *return*, the entire program is displayed; if by an *escape*, the program is displayed line by line for editing.

```
#Break (at) STEPesc
 ##Type @OLDCCcr
 ##Go (to) cr
 ##cr
<Program number is (2)>  #Break (at) cr
0123-0233 <R,W,X>  0241-0456 <R,W,X>  0712 <R,W,X>  01000 <X>(1)
TICK <event>  STEP <event>(2)
#Program-edit ?(program-number) (<ESC>-to-edit or <RETURN>-to-view)
#Program-edit 2cr
   Type @OLDCC
   Go (to)
#
```

When editing a line of a break-time program, the user can specify that the next (\) or prior (↑) line be displayed or that a replacement (R) of the current line or an insertion (I) in front of the current line be made. Editing is terminated by an empty editing specification. Replacement or insertion is identical to the specification of a break-time program within the break command in that a subcommand mode is entered where successive break-time commands can be entered until an empty command is supplied; then editing continues with the next line of the program. An extra (dummy) last line is added when editing a program so that new commands can be inserted at the end; the dummy line is discarded when the command is terminated.

```
#Program-edit 2esc
   Type @OLDCC   ;?(↑ <prior>) or (\ <next>) or ((I<nsert>) or (R<eplace>)
 (commands))
   Type @OLDCC   ;Replace
 ##Mode Instruction esc ###Type @OLDCCcr
 ##cr
   Go (to)   ;cr
#Program-edit 2cr
   Mode Instruction  ###Type @OLDCC
   Go (to)
#
```

Single-step. Transfers control to the target program through the program counter for execution of one instruction. The single coded character *line-feed* effects this command.

Display

The display commands permit the user to search or examine the contents of designated locations (and, in two cases, optionally permit their replacement) or to evaluate expressions. The commands are:

Type. Displays location and contents of a list of expression-ranges, permitting the contents of each location to be replaced if the list is terminated by an *escape*, as in the following example.

```
#Type ?((expression-range)-list) optional-<escape>-to-modify
#Type 0:2esc    00: 00 = 1cr
 01: 00 = 2cr
 02: 00 = 3cr
#
```

The replacement value can actually be a list of expressions, the values of the expression

in the list going into successive locations starting with the one last displayed. If no new value is supplied before the terminator, the existing value is not modified.

```
#Type 0:2esc    00: 01 = 2esc    01: 02 = esc    02: 03 = 1esc   #
```

In Display-with-replacement only, the coded characters *back-slash* and *up-arrow* can also serve as terminators and perform special functions: *back-slash* causes the next location to be displayed for replacement and *up-arrow* causes the prior location to be displayed for replacement; both of these terminator characters permit the user to step beyond the limits of the ranges entered as arguments to the Type command.

```
#Type 010esc    010: 00 = 1↑    07: 00 = 2\    010: 01 = 3\    011: 00 = ↑
 010: 03 = 4↑    07: 02 = 5↑    06: 00 = \    07: 05 = \    010: 04 = \
 011: 00 = 6\    012: 00 = 7cr
#
```

The last location displayed by a type command becomes the "open" location, and the location following the last one displayed or replaced becomes the "next" location (see the next four commands).

Same. Redisplays the "open" location (see the Type command). The single coded character ":" effects this command. The commands Same, Prior, and Next are all shown in the following example.

```
#:  02: 01   #↑  01: 02   #\  02: 01   #\  03: 00   #
```

Prior. Displays the location at one less than the "open" location (see the Type command). The single coded character *up-arrow* effects this command. See the examples under Type, Same, and Equals.

Next. Displays the "next" location (See the Type command; the mode in which the open location was last displayed determined how far it was advanced to the "next" locations.) The single coded character *back-slash* effects this command. See the examples under Type, Same, and Equals.

Equals. Displays the "open" location (see the Type command) as bits or as a number if the current output mode is already bits. The single coded character "=" effects this command. In the following example format *F2* has been declared consisting of four half-word fields.

```
#Mode Formatted F2cr
#:  010: 00,01,02,03   #=  010: 01   #\  011: 02,03,04,05   #\  013: 06,07,00,01
#↑  012: 04,05,06,07
```

Locate. Finds cells in a list of expression-ranges that contain (or do not contain) a specified value, examining only those bits designated by an optional mask, and displays their locations and contents, permitting each displayed value to be replaced if the list is terminated by an *escape*. The comparison value and mask are expressions terminated by an *escape*; the comparison value defaults to "NON 0" and the mask defaults to all 1's. The search is performed over a list of ranges, as for the Type command.

```
#Locate ?((expression) or NON (expression)) <match value defaults to NON 0>
#Locate NON 0esc (with mask) ?(optional-expression) <mask value>
#Locate NON 0 (with mask) esc<not zero> (in) ?((expression-range)-list)
 optional-<ESC> to-modify
#Locate NON 0 (with mask) <not zero> (in) 0:020cr
 00: 01    01: 02    02: 03    07: 05    010: 04    011: 06    012: 07
```

It is important that the comparison value, the mask, and the data be properly aligned. For example,

```
#Locate 070^esc (with mask) 070^esc (in) 0:31^cr
```

displays all cells from 0 through 31 whose second octal digit from the right contains all 1's.

When the command is terminated by an *escape* the debugger stops after each display to permit replacement, as for the Type command.

```
#Locate ^esc<non-zero> (with mask) 07^esc (in) 0:020^esc   00; 01 = 3^cr
017; 07 = ^cr
#
```

<u>Jump-history.</u> Displays the most recent target-program jumps in the order they occurred. The number of such jumps to display (taken modulo the default value) may be supplied.

```
#Jump-history ?((expression) or (empty <all>))
#Jump-history 3^cr
01000--0200(2 times)  0300--0100 #
```

<u>Evaluate.</u> Prints the value of a single expression. It has no effect on the open location and does not permit replacement.

```
#New-symbols PATCH^esc <at> 070000^cr
#Evaluate PATCH^esc = 070000   #
```

Storage

Storage commands change the contents of designated locations without displaying them and without changing the "open" location. The storage commands are

<u>Clear.</u> Clears the contents of a list of expression-ranges to all zero bits. Clearing an event for which a breakpoint has been established causes the event to be deactivated; it may be reactivated with a Set command. This may be of benefit when a break-time program has been associated with the event as the breakpoint data-base entry for that event is not affected.

```
#Clear 03^esc #
```

<u>Set.</u> Sets the contents of a list of expression-ranges to the value of an expression or (on default) to all one-bits. If the list is terminated by an *escape*, a single replacement expression is accepted; if it is terminated by a *return*, the default value of all 1's is used. The replacement expression is truncated to fit into the designated locations, if necessary. Setting an event for which a breakpoint has not been established (*i.e.*, for which there is no entry in the breakpoint data base) causes the event to be activated for a single occurrence of that event (with no break program associated), after which the event is automatically cleared.

```
#Set ?((expression-range)-list)
#Set 0.3^cr

#Set 0.3^esc = 2^cr
#
```

TARGET EXECUTION STATE

Target execution is initiated, or resumed, through explicit commands (exec Go, debugger Go or Single-step). Execution proceeds until a terminating event occurs, causing control to return to the appropriate PRIM command level. When execution terminates, the entire emulated context -- including clocks and outstanding IO operations -- is cleanly frozen until the next time execution is resumed. Except for explicit modifications to the context made by the user at the command level, the termination and subsequent resumption of execution is transparent to the target machine. The terminating events are

The target machine halts normally or is interrupted (by the emulator) due to the occurrence of some anomaly condition. A message to that effect is generated. The anomalies being monitored are listed in the tool-specific user guide.

The user enters an *abort*. The abort character is echoed and, after execution is stopped, a status message is output indicating the point of interruption.

The emulator detects the occurrence of a break condition established by the user via the debugger breakpoint command. The establishment of breakpoints and the subsequent interruption of execution at the time of their occurrence is the primary program debugging tool in PRIM.

An IO error occurs. A message detailing the particular device involved and the nature of the error is output. IO errors always return control to the exec state; the error messages and their meanings are listed at the end of this section.

When one of these conditions occurs, it is logged and execution continues until the end of the current cycle of the target emulator. It is therefore possible for multiple conditions to result in a single stop. When this is the case, the action and message appropriate to each of the conditions is produced.

When a breakpoint is detected, the debug program, if any, associated with each breakpoint is executed by the debugger before control returns to the command level. Should some break program terminate without a Go -- or should there be some break with no break program -- a message describing the break is output and the command level is entered. Otherwise, execution is automatically resumed; the user receives no indication that a breakpoint occurred unless the break program itself produced output.

TARGET I/O

The target machine that runs in PRIM consists of a processor (CPU) in some particular configuration built by the user to resemble the actual configuration required by his programs. A configuration is built -- before execution is begun -- by installing peripheral devices and establishing values for various machine options (see the exec Install and Set commands). After an emulated device has been installed, and before IO operations can proceed on that device, a (TENEX) file or assignable device must be associated with that emulated device (see the exec Mount command). Subsequent IO operations addressed to that device are then performed on the mounted file.

A mounted file may contain either direct device data (binary) or ASCII text; in the latter case, characters are translated between ASCII and the actual device character set as

they are processed. (If the device character set does not include lower case, input lower case letters are converted to upper case before translation.) When the target device is a record-oriented device (*e.g.*, card reader or punch) and the file is ASCII, then each record operation is performed on a line of the ASCII text file, including truncation and/or blank padding on input.

The mount option *THIS-TERMINAL* associates the user terminal (the one being used to communicate with PRIM) with a given device. When the terminal has been mounted on some device, then input from the terminal is switched between PRIM and the target machine every time execution is resumed and terminated. The intervention characters, however, retain their intervention meanings. To allow the full ASCII character set to be input to the target device from the terminal, there is a *control-shift* escape character defined during target execution. To help distinguish PRIM output from target output directed to *THIS-TERMINAL*, all PRIM-generated output is prefixed with the herald "--> " at the beginning of a new line. This applies in particular to both stopping messages and typeout resulting from break-time debugger programs.

I/O ERROR MESSAGES

Various I/O errors may occur. When any one occurs, execution -- including the error-generating operation -- is suspended, and control returns to the PRIM exec. When execution is next resumed, the suspended operation is retried unless it has been explicitly canceled by the user using the exec Cancel command.

"File not mounted."

The indicated device has no file mounted. If a file is mounted before execution is next resumed, the operation will be performed then. (An installed device to which no IO is directed need not have a mounted file in order to run.) The operation may instead be canceled.

This message is also produced when an output operation occurs on a device which has been mounted for input only, and vice versa. Again, a second file must be mounted on the appropriate side of the device in order to proceed normally with the program.

"File not open."

The indicated device has an inaccessible file mounted on it. The device must either be reassigned or unmounted and then mounted. The situation is similar to the case above, except for the possibility of reassigning.

"Improper tape format detected."

TENEX files which are mounted on target magnetic tape devices are encoded in a unique internal format that requires such files to be used only for PRIM magnetic tape devices. The mounted file is inconsistent with that format. The device must be unmounted and replaced with a proper tape file.

"Device not installed."

A device that is referenced by the program is not installed. Should the missing device be required, there is no way to continue this session, since device installation is no longer allowed. Should the reference be a mistake, execution may be continued down a different path (the operation will be automatically canceled when execution resumes).

"ASCII input character not recognized -- ignored."

The last character read from the ASCII input file on the designated device was not translatable into the character set of the device. The character has been skipped over; resuming execution causes the read operation to continue with the next character in the file. The position of the offending character in the file may be determined via the exec Filestatus command, specifying the indicated device.

Any other error indicates a bug either in the emulator or in PRIM. Such errors should be reported.